TRANSPUTER IMPLEMENTATION OF SIGNAL PROCESSING ALGORITHMS

A Thesis Submitted

in Partial Fulfilment of the Requirements

for the Degree of

MASTER OF TECHNOLOGY

by

GALGALI SHREENIVAS

to the

DEPARTMENT OF ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

July, 1992

CERTIFICATE

It is certified that the work contained in the thesis entitled "TRANSPUTER IMPLEMENTATION OF SIGNAL PROCESSING ALGORITHMS" has been carried out by Galgali Shreenivas under my supervision and that this work has not been submitted elsewhere for a degree.

20.7.92

(M.U.Siddiqi)

Department of Electrical Engineering

I.I.T. KANPUR.

July, 1992.

ABSTRACT

This thesis aims at implementing some signal processing algorithms on a transputer network. Matrix multiplication, convolution, discrete Fourier transform (Goertzel algorithm), fast Fourier transform (radix 2, radix 4), fast Walsh–Hadamard transform, two dimensional fast Fourier transform, and two dimensional convolution are selected for parallelization. Parallel algorithms are derived from their sequential versions by using partitioning approach. The parallel algorithms are implemented on a network of 2,4,8 transputers. All the program coding is done in Occam, the native language for transputers. The program development, testing, and debugging are done in the Transputer Development System (TDS) environment. The speedup factors and efficiencies of the parallel algorithms are measured for different sizes of input data.

EE-1992-M-SHR-TRA

TO
MY PARENTS

ACKNOWLEDGEMENT

I express with pleasure my deep sense of indebtedness to Prof. M U Siddiqi for suggesting to me this topic and for his valuable guidance throughout the work. He has been a constant source of inspiration and motivation. I am thankful to my course teachers Dr. V P Sinha, Dr.(Mrs.) S Gupta, Dr. A Mahanta, Dr. P R K Rao and Dr. R K Bansal.

My special thanks to Subbarao for his company, cooperation and suggestions.

I thank Hari, Madhu, Uday, Venkatesh, Deepak, Mahajan and Venu who helped during various stages of thesis work and preparation. My thanks to Alok, Pandey and Balwinder for their company and cooperation during the last phase of my work.

My stay at IIT Kanpur was a memorable one. I am grateful to Viju, Jaytheerth, Jagirdar, Arvind, Raghu, Anand, Tomar, Sanjeev, Sujeet, Pankaj and others for this.

I am grateful to Sudhanand, Ajaraj and Jayshree for there moral support and encouragement.

TABLE OF CONTENTS

Chapter 1
INTRODUCTION

1.1. Review of parallel processing:

Over the past four decades the computer industry has experienced four generations of development, physically marked by the rapid changing of building blocks from relays and vacuum tubes to discrete diodes and transistors, to small and medium scale integrated circuits and to large and very large scale integrated devices. Computer usage is gradually becoming more and more sophisticated. The increasing levels of sophistication are

- Data processing
- Information processing
- Knowledge processing
- Intelligence processing

Each level of increasing sophistication demands more powerful and faster computers.[1]. There are two methods to achieve higher performance .First,by increasing the speed of the circuitry. Second, by increasing the number of operations that can take place concurrently. The speed of light puts a ceiling on the speed at which electronic components of a certain size can operate. Hence now greater stress is given on the second method.

Parallel processing is a kind of processing that emphasizes the concurrent manipulation of data elements belonging to one or more processes solving a single problem. A *Parallel computer* is a computer designed for the purpose of parallel processing. The concept of parallel processing can be introduced at different levels of program execution like:

- Program level
- Task or procedure level
- Inter Instruction level
- Intra Instruction level

The highest level of parallel processing is conducted at the level of programs. This requires the development of parallel processable algorithms. Implementation of parallel algorithms depends on the efficient allocation of limited hardware–software resources to multiple programs being used to solve a large computation problem. The next higher level of parallel processing is conducted among procedures (program segments) within the same program. This involves decomposition of a program into multiple tasks. At the third level concurrency among multiple instructions is exploited. Finally, we wish to have faster and concurrent operations within each instruction. Parallelizing at the program level is often conducted algorithmically. Parallelizing at the intra instruction level is often implemented directly by hardware means. The role of hardware in implementing parallelism increases as we move from higher to lower levels. On the other hand, the role of software in implementations increases from lower to higher levels. In practical situations, there prevails a trade–off between the hardware and software approaches to solve a problem. Parallel processing is a combined field of study requiring a broad knowledge of, and experience with, all aspects of algorithms, languages, software and hardware. A number of hardware and software means have been developed to promote parallelism in *uniprocessor* computers.[1].

Hardware approaches:

1. Multiplicity of functional units:

Many of the functions of the ALU can be distributed to multiple and specialized functional units which can operate in parallel.

2. Parallelism and pipelining within the CPU:

Parallel adders are built into ALUs. Various phases of instructions are now pipelined, including fetch, decode, operand fetch, arithmetic logic execution, *etc.*

3. Overlapped CPU and I/O operations:

I/O operations can be performed simultaneously with CPU computations by using separate I/O controllers, channels,or I/O processors.

4. Use of a hierarchical memory system:

Usually, the CPU is faster than memory access. A hierarchical memory system can be used to close up the speed gap. Cache memory can be used to serve as a buffer between the CPU and the main memory.

Software approaches:

1. Multiprogramming:

The interleaving of CPU and I/O operations among several programs is called *multiprogramming*. Total execution time is reduced with multiprogramming.

2. Time sharing:

Timesharing is a kind of multiprogramming that assigns fixed or variable *time slices* to multiple programs.

Uniprocessor systems have their limit in achieving high performance. First, because of the physical limitation of the current semiconductor design and fabrication processes a single processor cannot be made dense enough to ensure very small time delays that mean high speed. Another limitation is called the *Von Neumann bottleneck*. This results from the use of a single memory interface through which all data and control information (i.e programs) must pass. A obvious solution is to replicate the best of what we have to achieve the level of capability we desire. Various ways to combine and operate processing elements

result in different types of architectures.[2]. Systems having multiple processors are mainly divided into

- Multiprocessors
- Array processors
- Vector processors

The three essential issues that must be considered for a parallel architecture are the *granularity* of the processing elements, the *topology* of the interconnections between processing elements, and the *distribution of control* across the processing elements. Granularity refers to the power of each processing element in the architecture. Topology refers to the pattern and density of the connections that exit between the processing elements. Control distribution is concerned with allocating tasks to processing elements and synchronizing their interactions.

A multiprocessor system contains two or more processors of comparable capabilities. All processors share access to a common set of memory modules and I/O devices and the entire system is controlled by a single integrated operating system controlling interactions among processors and their programs at various levels. Processors in a multiprocessor system may operate asynchronously. In the literature, the term multiprocessor is used to refer to a shared memory multiple CPU computer, and the term multicomputer is used to refer to a multiple CPU computer lacking shared memory, in which process cooperation occurs through message passing.

An array processor uses multiple synchronized processing elements (PEs). Each PE in the array is essentially a conventional computer. An appropriate data routing mechanism must be established among the PEs. An array computer contains a fairly powerful conventional computer as a controlling processor. Its functions include indicating what operations are to be performed by PEs during a cycle; enforcing an interconnection pattern among the PEs so that the array data is properly routed.

Vector processors perform the same operation on all elements of a vector at once. In order to perform this function, vector processors are usually attached to and run in conjunction with a scalar processor. The scalar processor is usually a conventional computer that handles the execution of scalar instructions in the instruction stream.

Architectural Classification Scheme of Flynn [3]:

Flynn's classification is based on the multiplicity of instruction streams and data streams in a computer system. An instruction stream is a sequence of instructions executed by a computer. A data stream is a sequence of data used to execute an instruction stream. The multiplicity is taken as the maximum possible number of simultaneous instructions (operations) or data (operands) being in the same phase of execution at the most constrained component of the organization.

Single Instruction stream–Single Data stream (SISD):

Computers in this category can decode only a single instruction in unit time. Most serial computers fall into this category.

Single Instruction stream–Multiple Data stream (SIMD):

Processor arrays fall into this category. A processor array executes a single stream of instructions, but contains a number of arithmetic processing units, each capable of fetching and manipulating its own data.

Multiple Instruction stream–Single Data stream (MISD):

No computers fit into this category.

Multiple Instruction stream–Multiple Data stream (MIMD):

This category contains most of the multiple processor systems. Processors execute different instruction streams on their own data.

Parallel Computing Terminology.[3]:

The *worst–case time complexity* (or simply *time* or *complexity*) of a parallel algorithm is a function $f(n)$ that is the maximum, over all inputs of size n, of the time elapsed from when the first processor begins execution of the algorithm until the last processor terminates algorithm execution. The *cost* of a parallel algorithm is defined as its complexity times the number of processors. For example, a sequential algorithm to determine the sum of n values has complexity $O(n)$, since it requires $n-1$ additions. If additions are allowed to be done in parallel and $n/2$ processors are available, then the sum can be determined in $\lceil \log n \rceil$ steps by computing partial sums. Thus, parallel addition has complexity $O(\log n)$ with $n/2$ processors.

Two important measures of the quality of parallel algorithms implemented on multiple processor system are ***speedup*** and ***efficiency***. The speedup achieved by a parallel algorithm running on p processors is the ratio of the time taken by the parallel computer executing the serial algorithm to the time taken by the same parallel computer executing the parallel algorithm using p processors. The efficiency of a parallel algorithm running on p processors is the speedup divided by p.

A parallel algorithm is said to exhibit *linear* speedup if the speedup with p processors is p. Speedup cannot be greater than linear because a sequential computer can always emulate a parallel computer. Parallel algorithms usually have associated overheads and the hence the speedup would be less than linear.

1.2 Scope of the work:

Nowadays signal processing is frequently used in different applications, and signal processing as used now requires to operate on large real time data. Hence in order to obtain meaningful results, it is desirable to have *high throughput* systems (Throughput of a system is the number of results it produces per unit time). Signal processing algorithms are highly structured and computationally intensive. So in order to have high throughput, these

algorithms are to be executed as fast as possible. This is done using a system having multiple processing elements and here we have selected *transputers* as processing elements. The reasons for using transputers as processing elements are; First, the transputers require minimum external hardware support for good interprocessor communication. Second, the transputer (being based on the RISC architecture) is very powerful and compact. Third, the network can be easily expanded in size, by adding additional transputers. Fourth, transputer has the support of *Occam* which is the native programming language for transputers.

In order to fully exploit the capabilities of parallel architectures proper algorithms are to be designed. Approaches like ***pipelining*** and ***partitioning*** have been proposed for designing parallel algorithms. Pipelining achieves concurrency by dividing a computation into a number of steps while partitioning is the use of multiple resources to achieve concurrency. In this thesis, we use partitioning approach. There are no fixed rules for designing parallel algorithms. More than standard procedures and techniques, the design of a good parallel algorithm has to rely upon intuition and experience.

The aim of this thesis is to implement some signal processing algorithms on a transputer network using Occam.

Implementation:

Phases in implementation of an algorithm are as follows.

1. Algorithm is parallelized from its sequential version.

2. The resultant algorithm is coded in occam as a number of communicating parallel processes and tested on a single processor machine simulating the effect of parallelism.

3. The algorithm is now run on a transputer network in which each transputer implements one or more of the Occam processes.

1.3. Organization of the thesis :

The organization of the thesis is as follows :

Chapter 2 presents a few general statements about parallel algorithms for MIMD computers.

Chapter 3 describes the architecture of transputer and its native language Occam. Key issues involved in the programming of transputer networks are discussed in this chapter.

Chapter 4 describes the hardware setup used for the implementation of algorithms

Chapter 5 describes sequential version of a few signal processing algorithms and their parallelization using partitioning approach.

Chapter 6 describes the performance of the algorithms developed in Chapter 5 on a network of 2,4,8 transputers. It gives the results in tabular forms. The thesis is concluded in this chapter discussing the scope for further work.

Chapter 2
PARALLEL ALGORITHMS FOR MIMD COMPUTERS

The capability of parallel computers can be fully exploited by designing suitable algorithms. Parallel algorithms can be designed in three ways, by detecting and exploiting any inherent parallelism in an existing sequential algorithm, by inventing a new parallel algorithm, or by adopting another parallel algorithm that solves a similar problem. First method is the most widely used. However,transforming a sequential algorithm to parallel form may not give satisfactory results. Some sequential algorithms have no parallelism, a parallel algorithm made from such a sequential algorithm will exhibit poor speedup.

Factors that must be taken into account while designing a parallel algorithm are [3]

Knowledge of the problem: The knowledge of the problem helps in parallelizing the sequential algorithm, because from the external appearance the algorithm may not be parallelizable. For example the addition of 4 integers. The sequential algorithm would be $[(a + b)+c]+d$. From the way in which this is written the sum of a and b must be found before c is added to it and so on, but we know externally that the addition can be done in parallel, so we can write the above expression as $(a+b)+(c+d)$. In this case first and second parentheses can be computed independently.

Communication complexity plays an important role. Sometimes communication complexity is higher than computational complexity, i.e more time is spent routing data among processors than actually manipulating the data,.and this results in an inefficient algorithm.

The performance of an algorithm can be different on different *architectures* of the

parallel computers. This is due to different communication overheads for different architectures. The algorithm designed for one particular architecture may not be efficient on another architecture.

Another factor is the *synchronization* of the computations of different processors. In some methods, called *synchronous*, processors must wait at predetermined points for the completion of certain computations or for the arrival of certain data. In other methods, called *asynchronous*, there is no need to wait at predetermined points.

2.1. Algorithms for MIMD computers:

The main issues for the programmer of MIMD computers, given a problem with certain amount of parallelism are:

- Task allocation; i.e the breakdown of the total workload among a number of processors so that they can work efficiently toward a solution.
- The mechanisms that allow processes to work together, and the expression of these mechanisms in a programming language.
- Scheduling of the processes on processors.

Algorithms on MIMD computers can be divided on the basis of task allocation into three categories: *pipelined algorithms*, *partitioned algorithms*, and *relaxed algorithms*.

Pipelined Algorithms:

A pipelined algorithm is an ordered set of segments in which the output of each segment is the input to its successor. The input to the algorithm serves as the input to the first segment, the output of the last segment is the output of the algorithm. All segments should produce results at the same rate, or else the slowest segment will become a

bottleneck. Some call it as *micropipelining*. A *systolic algorithm* is a special kind of pipelined algorithm, in which computations performed at each stage are identical and the flow of data can be in more than one direction.

Partitioned Algorithms:

Partitioning is the sharing of a computation, unlike pipelining in which processors assume different computational duties. A problem is divided into *subproblems* that are solved by individual processors. Solutions of subproblems are then combined to constitute the problem solution. This combining of solutions implies synchronization among the processors. For this reason partitioned algorithms are sometimes called *synchronized algorithms*. For example the addition of kp values on a multicomputer with p processors. Here p processes are created, one per processor, each processor adds a unique set of k values. When a process has completed its subtotal, it adds the subtotals to obtain the grand total.

Partitioned algorithms can be divided into two categories, *prescheduled algorithms* and *self–scheduled algorithms*. In prescheduled algorithms each process is allocated its share of the computation at compile time. Prescheduling is used when a computation consists of a large number of subtasks, each of a known complexity. In self–scheduled algorithms the work is not assigned to the processors until run time. Processes schedule themselves as the program executes. A global list of work to be done is kept, and when a process is without work, a task is removed from the list.

Relaxed Algorithms:

An algorithm that works without process synchronizations is said to be relaxed. All processors may be working toward the same goal (as in partitioning), or there may be some specialization of purpose (as in pipelining), but no processor ever has to wait for another processor to provide it with data. Some authors call these as *asynchronous algorithms*.

In real life problems the distinctions often become blurred, and a parallel algorithm may have features of all three types.

2.2. Factors that limit Speedup [3]:

A number of factors contribute to limit the speedup achievable by a parallel algorithm on a MIMD model.

1). A constraint is the size of the input problem. If there is not enough work to be done by the number of processors available, then any parallel algorithm would show constrained speedup. This phenomenon is called the *Amdahl effect.* As a general rule, speedup is a nondecreasing function of the problem size.

2). The number of process creations and synchronizations in a partitioned algorithm must be minimized. If more processes are created then synchronizations must be done frequently, and the overheads can be significant, reducing the speedup. In order to improve the speedup the *grain size* (i.e relative amount of work done between synchronizations) should be made as large as possible.

3). *Amdahl's law*: A small number of sequential operations can effectively limit the speedup of a parallel algorithm. Let f be the fraction of operations in a computation that must be performed sequentially, where $0 \leq f \leq 1$. The maximum speedup S achievable by a parallel computer with p processors performing the computation is

$$S \leq 1 / (f+((1-f)/p))$$

In the case of partitioned algorithms, if a portion of the algorithm must be executed sequentially by one of the p processors, then the remaining $p-1$ processors must wait for the sequential portion to complete before they resume. This is the limitation caused by the use of sequential code in partitioned algorithm.

4). Contention for a single resource limits the speedup, in the same way as the presence of sequential code. Contention between processors for data resources is called *software lockout*. If the instructions used by all the processors are kept in a shared memory, then processor contention for the memory puts a lower limit on the speedup.

5). *Minsky's conjecture*: It states that the speedup achievable by a parallel computer increases as the logarithm of the number of processing elements, thus making large scale parallelism unproductive.

The ideal situation is every processor in a multiprocessor to have a local memory where instructions, local variables, and constants are stored. Shared memory is accessed only to find the value of a variable that is shared among processes. In other words, shared memory is used only for communication and synchronization among processes.

2.3. Communication and Synchronization [3]:

In order for processes to work together, they must have the ability to communicate and synchronize. Communication is achieved either through shared variables or through message passing. The type of communication available is dictated by architecture. For example, multicomputers do not have any shared memory, hence they must communicate through message passing. Communication often leads to synchronization requirements. Synchronization has two uses; to constrain the ordering of events and to control interference. For example, consider a pipelined parallel algorithm in which one process is writing into a buffer and another process is reading from it. A process can not write into a full buffer, and a process can not read from an empty buffer. A mechanism to prevent these events from happening is an example of the use of synchronization. The second use of synchronization is to control interference. For example a process should be prevented from accessing the global value while another process is updating it.

Expressing Concurrency: A number of programming language constructs to express concurrency have been proposed. These constructs, were developed to express concurrency in multiprogrammed operating system, but they can be used in a parallel programming environment. A few constructs are Fork and Join, Cobegin and Coend.

Synchronizing with Shared Variables:

Mutual exclusion and *condition synchronization* are two frequently used types of synchronization on systems in which processes communicate via shared variables. A *critical section* is a sequence of statements that must be executed as an atomic operation. In the case of parallel sum finding algorithm, each process finds a partial sum and then executes the statement

$$global\_sum = global\_sum + local\ sum.$$

If no mechanism were used to ensure the atomocity of this statement, then the final result may not be correct. If a process A read the current value of global_sum and performed the addition, resulting in a new value, but did not store this result before process B read the value of global_sum then the outcome of the entire computation would be wrong. Hence this statement is an example of a critical section. Mutual exclusion refers to the mutually exclusive execution of critical sections. Condition synchronization is a method for delaying the continued execution of a process until some data object it shares with another process is in an appropriate state.

Synchronization through Message Passing:

Message passing is a form of communication, since a process receiving a message is receiving values from another process. Message passing is a form of synchronization also, since a message can be received only after it has been sent. In order to send a message, a

process executes a statement of the form

send *expression_list* to *destination_designator*.

The values of the expressions in the expression list are put into a message that is sent to the designated destination. In order to receive a message, a process executes a statement of the form

receive *variable_list* from *source*_designator.

When a process receives a message from the designated source, it assigns the values in the message to the variables list.

One way in which message passing schemes differ is the means for specifying the source and destination designators. The simplest method is for the source and destination designators to be the name of processes. This is called *direct naming*, and it is easy to implement and use. Another way to implement message passing is for source and destination designators to refer to *global names*, or *mailboxes*. Many processes send messages to a mailbox, and processes receive messages from that mailbox.

A second way in which message passing schemes differ is, when the source and destination designators are decided. If they are fixed at compile time, then programs may use channels that are known at compile time. This is known as *static channel naming*. Static channel naming implies that a process must have permanent access to a channel, even if it requires use of it for only a short period. *Dynamic channel naming* delays the computation of source and destination designators until run time.

A third way in which message passing schemes differ is whether sending or receiving a message can delay the further execution of the process performing the statement. A *nonblocking* statement never delays the further execution of the invoking process. Otherwise, the statement is said to be *blocking*. If there is no buffer between a process sending a message and a process receiving a message, then both *send* and *receive* are

blocking: send delays until a corresponding receive is executed. At that time the message is passed, and then both sending and receiving processes continue execution. This is called *synchronous message passing.* If buffers are used to store messages that have been sent but not received (*buffered message passing*), then the system designer has a number of options. If a send is executed when the buffer is full, the send might delay until there is room in the buffer, or a condition code could be returned to the sender that the message could not be sent, owing to the full buffer. Similarly, if a receive is executed when the buffer is empty, then either it delays until there is a message in the buffer or it returns a condition code that the buffer was empty. If the buffer between a sending process and a receiving process is effectively unbounded, then send is nonblocking. This is called *asynchronous message passing* or *send no wait.* Asynchronous message passing allows the sending process to get arbitrarily far ahead of the receiving process.

2.4. Concurrent Programming Languages [3]:

Programming languages allowing concurrency can be divided into three categories: *Procedure–oriented languages, Message–oriented languages,* and *Operation–oriented languages.* It is possible that any category of concurrent programming language can be implemented on any kind of multicomputers or multiprocessors, but implementing a language that does not match the architecture will not help the programmer to develop efficient parallel algorithms.

In procedure–oriented languages, process interaction is based on shared variables. Processes have direct access to the data they want to manipulate. Hence concurrent access to shared data is possible. Concurrent Pascal, Modula are examples of this kind of languages. Since processors manipulate data directly, this kind of language is most suitable for multiprocessors.

Message–oriented languages are built upon the primitives send and receive. These do not give access to every data object. Every object has a caretaker process, which manages it. In order to manipulate an object, a process must send a message to its caretaker. Occam, Gypsy are examples of this kind of language. These are suitable for multicomputers. When it is used on a multicomputer, it makes the communication network transparent, simplifying the programmer's job.

Operation–oriented languages combine aspects of other two categories. Operations are performed on objects by calling procedures, as in procedure–oriented languages, but objects are managed by caretakers as in message–oriented languages. Mod, Ada are examples of this kind of languages. These languages can be implemented on multiprocessors and multicomputers.

Chapter 3
TRANSPUTER AND OCCAM

3.1. Transputer [4]:

"Transputer" can be used as a basic element in multiple processor systems. The word ***transputer*** may be interpreted as a contraction of the words ***tranceiver*** and ***computer***. The interpretation suggests that a transputer consists of a communication system and a computational element.

There are a number of interesting features of transputers which distinguish them from any other processors. The integration of the design and the advantage of having many features built into a single chip are the important aspects, which result in the enhancement of processor speed, reduced chip size, and simplicity of the system design.

A transputer can be used in a single processor system or in network to build high performance concurrent systems. A typical member of transputer product family is a single chip containing processor, memory and point–to–point communication links. A network of transputers can be easily constructed using these links. A variety of configurations can be built by hard–wiring transputers together, limited only by the number of links provided on each transputer. The current transputer systems have four to six links. Four links are enough to allow different useful configurations.

Scalability is a major advantage of transputer–based systems: it is much easier to enhance a system by adding additional transputers to it than would be the case with the systems having other processors. ***Compatibility*** – the ease of replacing one transputer model with another without major design changes in a system is also valuable. This extends to mixing models of transputer within one system.

Architectural details:

Transputers are RISC processors. The computational instructions follow RISC principles closely, and they attain the benefits claimed for RISC architecture. However, they also have a small number of important non–RISC instructions concerned with scheduling and message passing.

Transputer is unusual in its ability to execute many software processes at the same time. A program can be run on a transputer, in which case the concurrency of the processes will be simulated by hardware with no software intervention. The same program can also be distributed over several processors, provided the communication between the subprocesses is not too complicated. In this case the component processes will be run in real concurrency. Interprocess message passing and the necessary synchronization are achieved in hardware, and no operating system is needed.

Transputer has many inbuilt features. It has instruction processor, small amount of on chip memory, memory controller, DMA control for four independent fast links, a microcoded multitasking kernel, and a clock (Fig. 3.1). Commercially available transputers are the T212, T414 and T800 marketed by INMOS. The T212 is a 16–bit processor, the other two are 32–bit processors. The T800 has an on chip IEEE floating–point processor.

Processor:

The processor portion of a transputer is a traditional microprocessor. The processor normally obtains its instructions and data from the internal 4K RAM .Data and instructions can also be obtained from the links. The processor provides 32–bit addressing, memory is addressed byte–wise and stored in 4–byte units. Software sees no difference between on–chip and external memory except in speed.

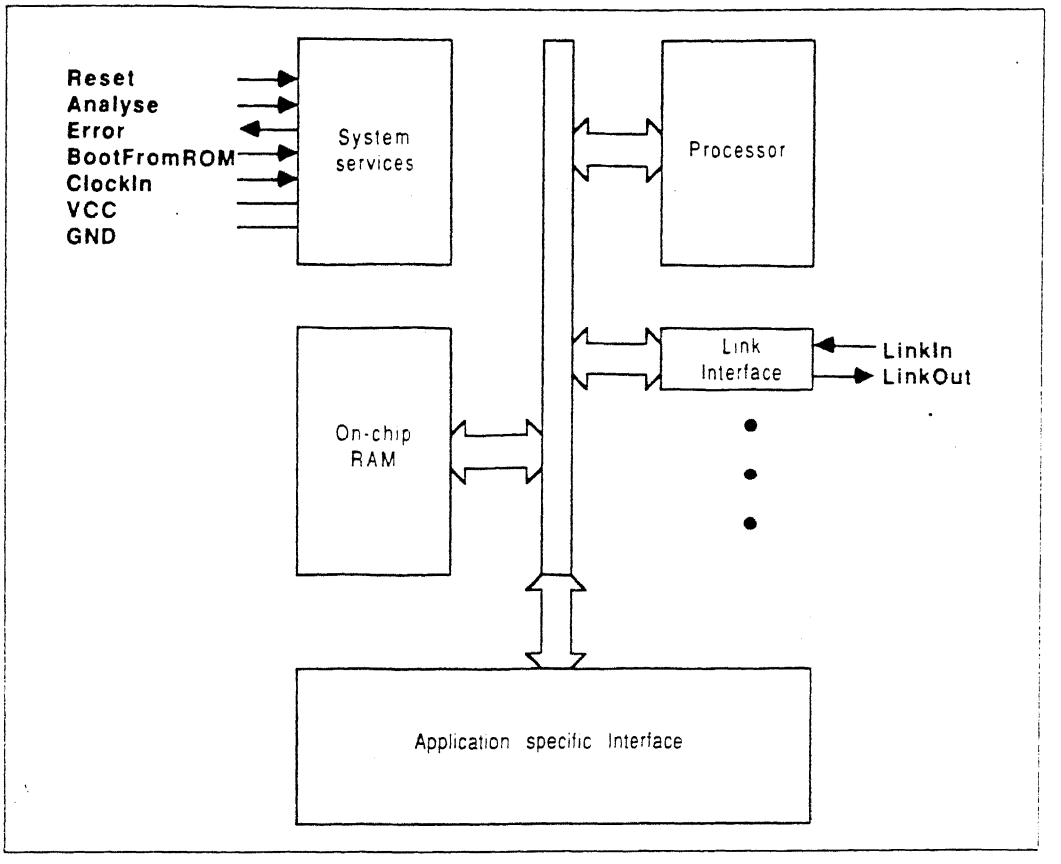

Fig.3.1: Transputer Architecture.

The design of transputer processor exploits the availability of fast on–chip memory by having only a small number of registers, the CPU contains six registers which are used in the execution of a process. The small number of registers, together with the simplicity of the instruction set enables the processor to have relatively simple (and fast) data–paths and control logic.

The registers are:

- The workspace pointer which points to an area of store where local variables are kept.
- The instruction pointer which points to the next instruction to be executed.
- The operand register which is used in the formation of instruction operands.
- A short, internal three register stack for expression evaluation (integer and address arithmetic).

The instructions refer to the stack implicitly. The use of a stack removes the need for instructions to respecify the location of their operands. The stack need not be saved when rescheduling occurs. Statistics gathered from a large number of programs show that three registers provide an effective balance between code compactness and implementation complexity.

Instruction set:

Transputer instruction set is designed for simple and efficient compilation. All the instructions have the same format. The instruction size of a transputer is only 8 bits. There are prefix instructions which allow the operand to be extended to any length. Measurements show that about 70% of the executed instructions are encoded in a single byte. Short instructions improve the effectiveness of the instruction prefetch, which in turn improves the processor performance. There is an extra word of prefetch buffer, so the

processor rarely has to wait for an instruction fetch before proceeding. Since the buffer is short, there is little time penalty when a jump instruction causes the buffer contents to be discarded. Also, the instruction set is independent of processor word length, allowing the same microcode to be used for transputers with different word lengths.

Memory controller:

The memory controller can drive external dynamic RAM with no additional circuitry. Together with the controller the processor can address a linear address space of 4 Gbytes. The 32–bit wide memory interface uses multiplexed data and address lines and provides a data rate of up to 4 bytes every 100 nanoseconds (40 Mbytes/sec) for a 30 MHz device. The configurable memory controller provides all timing, control and DRAM refresh signals for a wide variety of mixed memory systems.

Process scheduler:

The processor provides efficient support for concurrency and communication. It has a microcoded scheduler which enables any number of concurrent processes to be executed together, sharing the processor time. This removes the need for a software kernel. The processor does not need to support the dynamic allocation of the storage as the compiler is able to perform the allocation of space to the concurrent processes.

A process starts, performs a number of actions, and then terminates. Typically a process is a sequence of instructions. A transputer can run several processes concurrently. Processes may be assigned either high or low priority. At any time, a concurrent process may be:

active — being executed

— on a list waiting to be executed

inactive	–	ready to input
	–	ready to output
	–	waiting until a specified time

The scheduler operates in such a way that inactive processes do not consume any processor time. The active processes waiting to be executed are held on a list. This is a linked list of process workspaces, implemented using two registers, one of which points to the first process on the list, the other to the last (Fig. 3.2).

The IMS T800 supports two levels of priority. Priority 1 (low priority) processes are executed whenever there are no active priority 0 (high priority) processes. High priority processes are expected to execute for a short time.

Each process runs until it has completed its action, and is descheduled while waiting for communication from another process or transputer, or for a time delay to complete. Whenever a process is unable to proceed, its instruction pointer is saved in the processor workspace and the next process is taken from the list. Process scheduling pointers are updated by instructions which cause scheduling operations. Actual process switch times are less than 1 μs, as little state needs to be saved and it is not necessary to save the evaluation stack on rescheduling.

The processor provides a number of special operations to support the process model, including *start* process and *end* process. When a main process executes a parallel construct, start process instructions are used to create the necessary additional concurrent processes. A start process instruction creates a new process by adding a new workspace to the end of the scheduling list, enabling the new concurrent process to be executed together with the ones already being executed. When a process is made active it is always added to the end of the list, and thus cannot pre–empt processes already on the same list. The correct termination of a parallel construct is assured by the use of the end process instruction. This uses a workspace location as a counter of the parallel construct components which have still

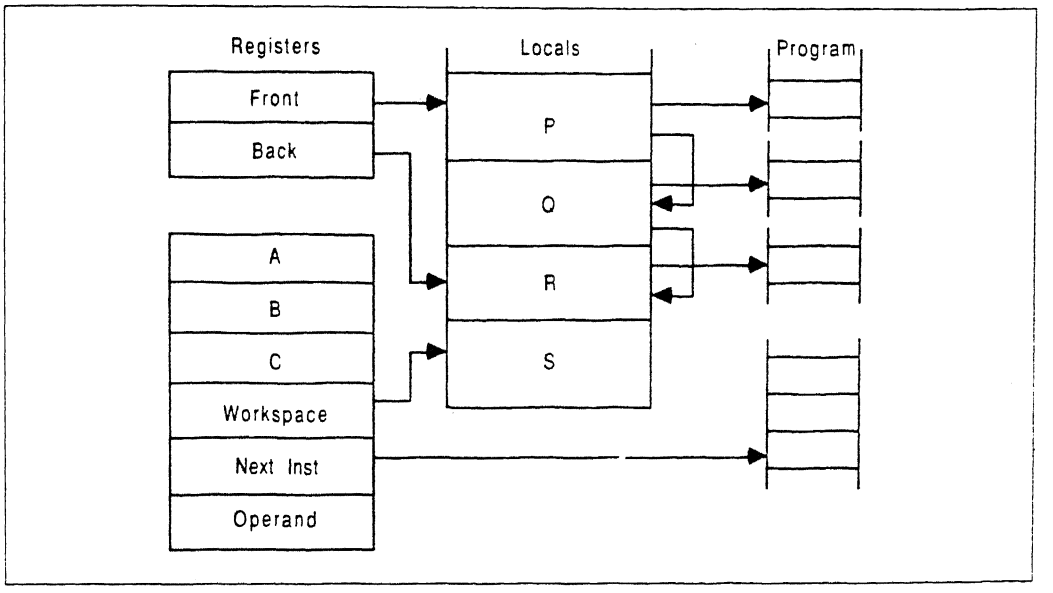

Fig.3.2: Linked Process List.

to terminate. The counter is initialized to the number of components before the processes are started. Each component ends with an end process instruction which decrements and tests the counter. For all but the last component, the counter is non zero and the component is descheduled. For the last component, the counter is zero and the main process continues.

Communications:

Communication between the processes is achieved by means of channels. The communication is point–to–point, synchronized and unbuffered. As a result, a channel needs no process queue, and no message buffer.

A channel between two processes executing on the same transputer is implemented by a single word in memory, a channel between processes executing on different transputers is implemented by point–to–point links. The processor provides a number of operations to support message passing, the most important being ***input message*** and ***output message***.

The input message and output message instructions use the address of the channel to determine whether the channel is internal or external. This means that the same instruction sequence can be used both for hard and soft channels, allowing a process to be written and compiled without the knowledge of where its channels are connected.

The communication takes place when both the inputting and outputting processes are ready. Consequently, the process which first becomes ready must wait until the second one is also ready. A process performs an input or output by loading the evaluation stack with a pointer to a message, the address of a channel, and a count of the number of bytes to be transferred, and then executing an input message or an output message instruction.

Communication links:

A link between two transputers is implemented by connecting a link interface on one transputer to a link interface on the other transputer by two one–directional signal

wires, along which data is transmitted serially. The two wires provide two channels, one in each direction. This requires a simple protocol to multiplex data and control information. Messages are transmitted as a sequence of bytes, each of which are to be acknowledged before the next is transmitted. A byte of data is transmitted as a start bit followed by a one bit followed by eight bits of data followed by a stop bit. An acknowledgement is transmitted as a start bit followed by a stop bit. An acknowledgement indicates that a process was able to receive the data byte and that it is able to buffer another byte.

The protocol permits an acknowledgement to be generated as soon as the receiver has identified a data packet. In this way the acknowledgement can be received by the transmitter before all of the data packets have been transmitted and the transmitter can transmit the next data packet immediately. The IMS T414 transputer does not implement this overlapping and achieves a data rate of 0.8 Mbytes per second using a link to transfer in one direction. However, by implementing sufficient overlapping and including sufficient buffering in the link hardware, the IMS T800 achieves a data rate of 1.8 Mbytes per second in one direction.

Timer:

Transputers have two timer clocks which 'tick' periodically. The timers provide accurate process timing, allowing processes to be descheduled until a specific time.
In the IMS T800 transputer, one timer is accessible to only high priority processes and is incremented every microsecond, cycling completely in approximately 4295 milliseconds. The other is accessible only to a low priority process and is incremented every 64 microseconds, giving exactly 15625 ticks in one second. It has a full period of approximately 76 hours.

Floating point processor:

The IMS T800 has an on chip IEEE floating point processor. The FPU operates concurrently with the CPU. This means that it is possible to do address calculation in the CPU while the FPU performs the floating point calculation. This can lead to significant performance improvements in real applications which access arrays heavily. Performance depends on many things, including clock and memory speeds, for the 20 MHz T800 these figures are of the order of 10 MIPS and 1MFLOPS.

As can be understood from above, all the links can be active at the same time as well as the processors. Thus a transputer can support nine truly concurrent activities. (one link can transfer data in both directions, of course, memory accesses have to be interleaved). On T800, the floating point processor operates in parallel with the instruction processor, which gives a tenth level of concurrency at the hardware level (but both the processors are controlled by a single instruction stream).

3.2. Occam. [5,6]:

Occam's Razor — ***Entities are not to be multiplied beyond necessity***, was the philosophy of the original implementation of Occam.

Occam is the native language for transputers. The design of Occam has been heavily influenced by the work on *Communicating Sequential Processes* (CSP), which gives a mathematical frame work for specifying the behaviour of parallel processes. Occam is based on the CSP model of computation with features chosen to ensure efficiency of implementation. In this model, an application is decomposed into a collection of communicating processes, and the processes communicate by passing messages.

Occam model is based on the idea of *process*. The software building block is a *process*. A system is designed in terms of an interconnected set of processes. Each *process* can be regarded as an independent unit of design. Its internal design is hidden and is

completely specified by the message it sends and receives. Internally, each process can be designed as a set of communicating processes. The system design is therefore hierarchically structured. Occam processes do not share any variables. Messages pass from exactly one process to the other. There are no multiple senders or receivers, no broadcasting, and no uncertainty about the origin of a message or where it is going. Messages are unbuffered, so sending and receiving a message involves momentary synchronization between the two participating processes. Messages are sent through static channels, as if through a circuit switched (rather than packet switched) network.

The maximum benefit from the transputer architecture can be obtained by programming the whole system in Occam. This provides all the advantages of a high level language, the maximum program efficiency and the ability to use the special features of transputers. The Occam model of concurrency is applicable equally to processes running on separate processors and to processes running on a single processor. Since the processor can be controlled only by one instruction stream, it is evident that ***the processes in one processor cannot be truly concurrent.*** However, the processes can be multiprogrammed, so that the effect of concurrency is reproduced. This 'similar concurrency' as distinct from 'real concurrency' is known as '***gratutious concurrency***'.

Occam provides a framework for designing concurrent systems using transputers just in the same way that boolean algebra provides a framework for designing electronic systems from logic gates. The system designer's task is eased because of the architectural relationship between Occam and transputer. A program running in a transputer is formally equivalent to an Occam process, and so a network of transputers can be described directly as an Occam program. Where it is required to exploit concurrency, but still to use standard languages, Occam can be used as a harness to link modules written in selected languages.

3.3. Programming transputers [7]:

The following are the points that concern the programmer of a concurrent system, designed using transputers.

Topology:

The pattern in which the processors are connected together is known as ***topology*** or the *configuration*. The idea of 'configuration' depends on the assumption that processors are connected permanently, or at least for the life of a whole program, an assumption which is true for current transputer systems. It is the responsibility of the designer of the overall system to decide how to configure the processors within it. Designing a topology for a large system can be difficult. It can sometimes be guided by an obvious mapping of a problem onto separate processors.

Placement:

Describing systems in terms of Occam processes allows algorithmic issues to be separated from the question of what hardware is going to perform those activities. This is a useful abstraction. One would like to be able to express an algorithm in the form of a program which is independent of hardware, so that it could be subsequently be performed using many different networks of processors. Each implementation would need a specification of how many processors were needed, how they were to be connected, and which processes were to be installed on which processors. This specification is called ***placement***. Transputer programs are not completely independent of their placement. A program will only run on one particular network of transputers. To run it on other networks, the program itself will have to be changed.

Non–determinacy:

Sequential programmers are used to the idea of a ***bug***. There are solid bugs and intermittent bugs. Intermittent bugs are data dependent, a program run on the same inputs will work every time or it may fail every time. Concurrent programming has a third type of bug, the bug that depends on the relative timing of concurrent processes. These are very often not repeatable, even if the program is rerun on the same data.

The problem arises because all the processors are allowed to run at their own speed. There is no attempt to constrain the processors into a lock step. Thus the order of events can change from one test run to another. Occam is precise about what individual processes do, but it cannot specify the relative timing of concurrent processes. It is the programmer's responsibility to ensure that when a program terminates it has completed the required function, regardless of the order in which things happened between starting and termination.

Deadlock:

A classic problem in concurrent systems is ***deadlock***. One part of a program is waiting for another to do something; the other is waiting for the first to do something else, and since both are waiting, neither can do what other expects. There may be a set of processes involved, rather than a pair. All transputer deadlocks are essentially the same in that they involve a closed chain of processes, each trying to communicate with another, but with no pair of them willing to participate in any one communication.

Software development [8]:

Transputer software is mostly developed under the '***Transputer development environment***' (TDS) supplied by INMOS.

TDS and related systems provide an integrated environment for editing, compiling and running the programs. They are centered on the '*folding editor*' which embodies an elegant and general way of representing and handling large amount of Occam text within a small screen. The TDS is so much an '*integrated environment*' that its files are not easily handled in other systems, not even in systems such as MS–DOS which is acting as the host or file server for the TDS. This makes it hard for the utility software on the host system to provide the facilities that are missing from the TDS.

Chapter 4
SETUP OF A TRANSPUTER NETWORK SYSTEM

4.1. General System Organization of a Processor Network [9]:

The overall configuration of a processor network consists of the following major components:

- Host computer
- Interface unit
- Processing Element array(s)
- Interconnection network(s)

Host Computer:

Generally, processing element array is used as an additional resource by a general purpose host running an operating system. PEs can be accessed by a procedure call on the host, or through an interactive, programmable command interpreter. The host computer is intended to provide system monitoring, data storage and management. It generates global control codes and object codes of PEs. It can be a microcomputer, workstation, minicomputer, main frame. The selection depends on the desired application.

Interface unit:

Interface unit is an interface between the host and PE array. Interface unit, connected to the host via host bus, or DMA, has the function of down loading, up loading, buffering array data and handling interrupts. The major function of the interface unit is to

support high bandwidth communication (accompanying high–speed processing) between the array and the host. By providing sufficient buffering, the unit must be able to balance the low bandwidth of the system I/O and the high bandwidth of the processor array.

PE arrays:

A PE array comprises of a number of processing elements with local memory. By providing sufficient local memory (both on chip and off–chip), PEs can effectively utilize its temporary data storage thereby saving communication time and avoid tying up of the interface bus.

Interconnection Network:

Interconnections within PE array are provided by the interconnection networks. These networks are generally large switching networks to provide flexibility of interconnections and are intended to provide high speed communications between the processing elements.

4.2. Hardware setup of the transputer network system [4,8]:

A nine–node transputer network has been set up in the image processing lab at IIT Kanpur (Fig.4.1). The setup contains two IMS B003 evaluation boards, each having four transputers and an IMS B004 transputer system development board having a transputer. Of these nine transputers (INMOS, UK), five are T800s and four are T414s. The host is an IBM PC–AT (80386 based). One of the transputers (T800 type) functions as the interface unit, known as ***Root node***, with 4 Mbytes of on board RAM. The host is connected only to the root node via a ***Link adapter*** (IMS C012). The root node performs the I/O between the processor array and the host. PEs are individual transputers each with 256 Kbytes of RAM on board. Transputer links are interconnected by a programmable switch known as ***Link switch*** (IMS C004).

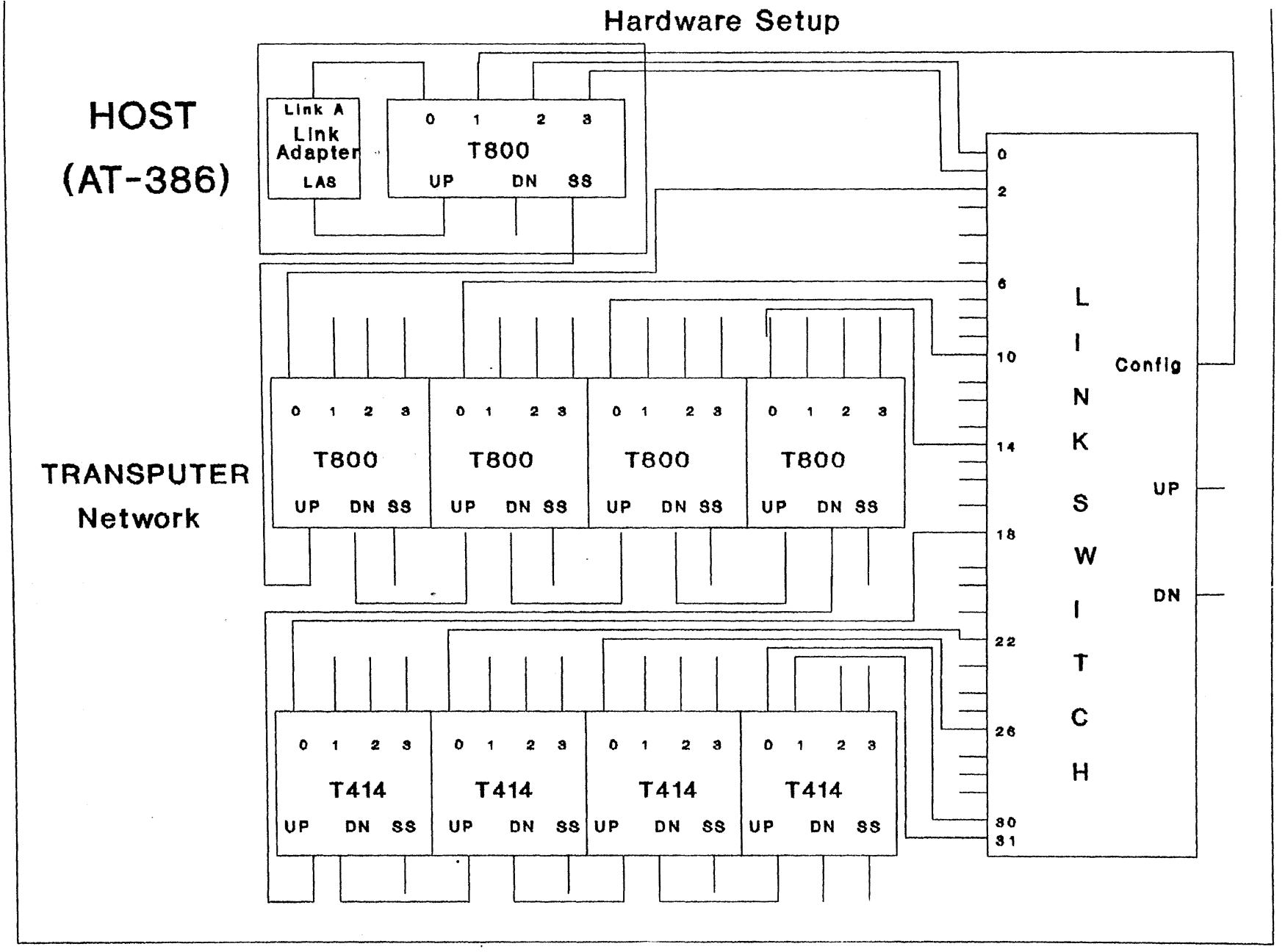

Fig.4.1: Hardware Setup

Link adapter:

Link adapter connecting the root node and the host provides for full duplex transputer link communication by converting bi–directional serial link data into parallel data streams.

Host:

The host functions as a file server for the system as memory management and file system are not supported by transputers. The server reads a DOS file to determine the network configuration (the configuration is specified at the compile time), the programs to be loaded and the boot order. The host loads the network via the ***Root transputer*** by sending loading information to it. The root transputer will boot the transputers connected to it, and route loading information to them, these will in turn boot and load other transputers in the network, until the whole network has been booted and loaded. The topology of the actual physical network should match that described in the program configuration description, otherwise the loading will fail.

Interface unit:

The root transputer acts as an interface unit between the host and the network. All communication between the host and individual transputers takes place through the root transputer. Most of the times the root transputer will be used for executing the I/O routines required for the host file server. Unlike dedicated interfaces, this can also share the computation load with other transputers.

Booting of the transputer network:

A communication protocol exists between the host transputer and the target transputer network to direct the code to the desired place in each transputer. The

communication consists of bootstrap packets, routing information, address information, code packets and execute items.

The bootstrap code for each transputer in the network is sent first. After all transputers in the network are booted, the code of each of the procedures allocated to processors in the configuration description is exported to the network preceded by the necessary routing and loading information. Following this, the code which calls the procedures (the main body) generated by the configurer is sent to each processor in turn and then each processor is told to start executing the loaded program.

Network Monitor:

INMOS boards provide system control functions to monitor and control the state of the transputer network. The system control connections on board are chained together to allow the whole of the network to be controlled from the host. The control connection consists of three signals.

- Reset: This is a signal from the host transputer to the network, which will reset all transputers in the network, ready for loading.
- Analyze: This is a signal from the host transputer to the network, which brings all transputers in the network to a controlled halt, so that their state can be examined.
- Error: This is a signal from the network to the host transputer, indicating that one of the transputers in the network has set its error flag.

Link switch:

Transputer network is interconnected using ***Link switch*** (IMS C004). It is a programmable link switch designed to provide a full crossbar switch between 32 link inputs and 32 link outputs. It introduces on the average only a 1.75 bit time delay on the signal. Link switches can be cascaded to any depth without loss of signal integrity and can be used

to construct configurable networks of arbitrary size. The switch is programmed via a separate link called the ***configuration link***.

In the setup, LINK0 of the root transputer is connected to the host (via link adapter) and LINK1 is connected is used as the configuration link for the link switch. So, only LINK2 and LINK3 on the root transputer are free. Also, since the link switch supports only 32 link connections, two links of the last (9th) transputer cannot be used at present (only LINK0 and LINK1 can be used on the last transputer).

4.3. Network Configuration:

Two transputer network:

This network is configured as shown in Fig.4.2(a), as this is the only possible configuration two transputers.

Four, Eight transputer network:

Different configurations of network are possible with 4 and 8 transputers. For all algorithms except for convolution we configured the networks as shown in Fig.4.2(b) and Fig.4.2(c), respectively, as communication overheads associated with these are less compared to other configurations. For convolution we configured the network as linear array.

4.4. Code development [8]:

Occam is used for the program development under the Transputer Development System (TDS) environment. The program development has two phases. In the first phase, the partitioned program is coded in Occam and tested as a parallel program of n processes running on a single transputer using soft channels for communication between the parallel processes. The TDS system has extensive library support even for most of the low level applications which can be used for simplifying the programming task.

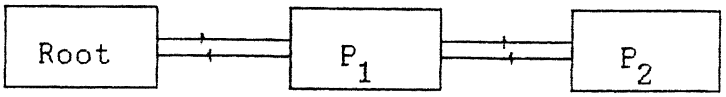

Fig.4.2(a): Two transputer network

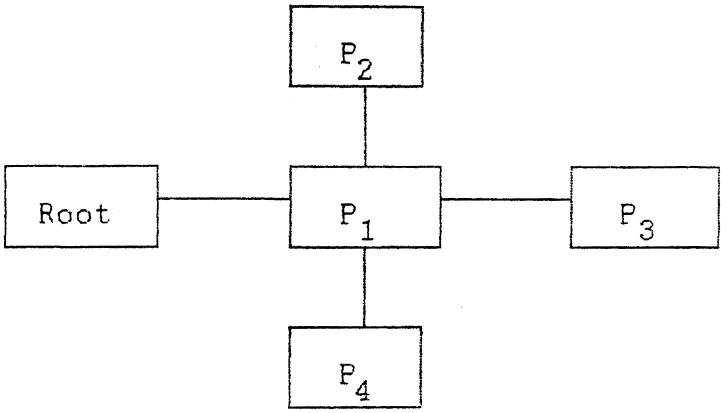

Fig.4.2(b): Four transputer network

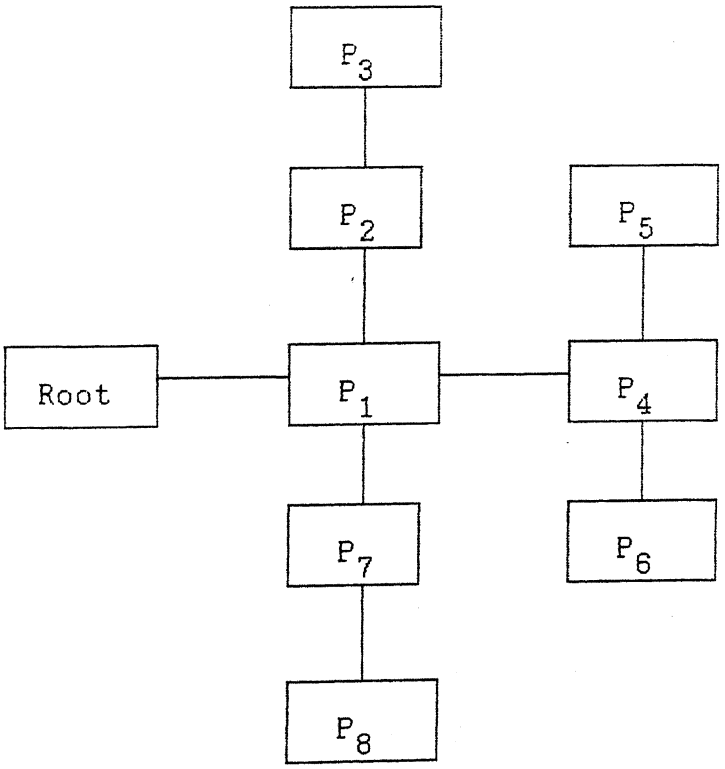

Fig.4.2(c): Eight transputer network

In the second phase, the same program is run on a processor network with the following changes made to the above program.

1). The individual processes are to be defined as procedures whose parameters should be only the hard channels and a processor identification number (sharing of variables between different processors is not possible) and should be compiled separately. The global constants can be shared by using libraries.

2). A mapping of the Occam channels to the transputer hard links should be given. This gives the network configuration information.

3). The separately compiled procedures are 'PLACED' on individual transputers and the program is compiled to generate a network code file.

For running the program, network link switch should be configured. This can be done by a software program which will transmit appropriate code to the configuration input link of the link switch.

The multi node program can be tested from the TDS environment. After successful run, it can be made a standalone program bootable by the external host by using the alien file server library routines available in the TDS environment.

Chapter 5
DESIGN OF PARALLEL ALGORITHMS FOR A TRANSPUTER NETWORK

Transputers are configured in a network using point to point serial links and communication is achieved through message passing. Now we shall consider the development of some specific algorithms for a transputer network. Here we detect inherent parallelism in existing sequential algorithms and then modify them into parallel algorithms. Partitioning technique is used to parallelize the algorithms. The algorithms developed here are prescheduled. We assume that p processors are available where p is less than N, the size of the input problem, and it is assumed that N is divisible by p.

5.1. Matrix multiplication [3]:

The product of $N \times M$ matrix A and $M \times L$ matrix B is $N \times L$ matrix C whose elements are defined by

$$c_{i,j} = \sum_{k=0}^{M-1} a_{i,k}\, b_{k,j}$$

The above equation can be written in the recursive form as

i, From 0 For N

  j, From 0 For L

    k,From 0 For M

$$c_{i,j} = c_{i,j} + a_{i,k} \times b_{k,j} \qquad (5.1)$$

Steps in the implementation of the sequential algorithm are:

1. Initialize the parameters.
2.
    a. Start the computation for i=0.
    b. Compute Eq.5.1 for j=0 and for all values of k.
    c. Repeat step 2.b for all values of j.
3. Repeat step 2 for all values of i.
4. Store the results.

Computation of Eq.5.1 for i=r and j=q does not depend on the the computation of Eq 5.1 for i=r−1 and j=q−1, i.e all.elements of C can be calculated independently. The inputs to the computation of Eq 5.1 for i=r and j=q are A(r,0), A(r,1),...,A(r,M−1) and B. Hence C(r,q) for different values of r can be calculated concurrently on different processors. Hence the algorithm for any processor in a transputer network can be given as:

1. Receive $N/p \times M$ elements of A and $M \times L$ elements of B.
2. Compute Eq.5.1 for the received values.
3. Send the values.

The root processor reads input values and sends them to the first processor. The first processor sends them to all processors, which compute and send back computed values to the first processor, which in turn sends them to the root processor.

5.2. Convolution [10,11]:

Consider two signals $x(t)$ and $h(t)$. The convolution of $x(t)$ and $h(t)$ yields the signal $y(t)$ which is defined as

$$y(t) = \int_{-\infty}^{\infty} x(\tau)\, h(t-\tau)\, d\tau = x(t)*h(t)$$

For finite duration discrete sequences $x(n)$ of length N, and $h(n)$ of length L, the convolution is given by

$$y(i) = \sum_{k=0}^{N-1} x(k)\, h(i-k) \text{ where } 0 \leq i \leq (N+L-2) \qquad (5.2)$$

$y(n)$ is a finite duration sequence of length $(N+L-1)$. The above equation can be written in the recursive form as

i, From 0 For N

j, From 0 For L

$y[i+j] = y[i+j] + [x(i) \times h(j)]$

Each iteration of the above equation yields partial sums, which are added together to obtain the convolution.

Steps in the implementation of the sequential algorithm are:

1. Initialize the parameters.
2.
   a. Start the computation for $i=0$.
   b. Compute Eq 5.2 for $j=0$.
   c. Repeat step 2.b for all values of j.
3. Repeat step 2 for all values of i.
4. Store the results.

Computation of Eq 5.2 for $i=k$ does not depend on the the computation of Eq 5.2 for $i=k-1$, except for the addition of the partial sums. The inputs to the computation of Eq 5.2 for $i=k$ are $x(k)$ and $h(0),h(1),\ldots,h(L-1)$. Hence $y(i)$ for different values of i can be calculated concurrently on different processors. Hence the algorithm for any processor in a transputer network can be given as:

1. Receive N/p samples of x and L samples of h.
2. Compute Eq.5.2 for the received values.
3. Receive the values computed in the other processors.
4. Add the values computed in this processor with the values obtained from the other processors.
5. Send the values.

The root processor reads input values and sends them to the first processor. The first processor sends them to all processors, which compute and send back computed values to the first processor, which in turn sends them to the root processor.

5.3. Discrete Fourier transform [11]:

The Fourier integrals provide the means for obtaining the frequency domain representation (the *spectrum*) of a signal from its time domain representation and vice – versa, i.e

$$\text{Fourier transform : } X(w) = \int_{-\infty}^{\infty} x(t).exp(-jwt)\ dt$$

$$\text{Inverse transform : } x(t) = \int_{-\infty}^{\infty} X(w)\ exp(jwt).dw$$

where $x(t)$ is the time domain signal and $X(w)$ is the frequency spectrum of $x(t)$. The above pair can be made suitable for digital computation by sampling the time domain and frequency domain variables and limiting the computation to a finite set of data points. The modified version of the Fourier transform is referred to as the *discrete Fourier transform* (DFT).

The DFT pair of a sequence $x(n)$, $n = 0,1,2,...,N-1$, is defined as

DFT : $$X(k) = \sum_{n=0}^{N-1} x(n)\, W_N^{nk} \qquad k = 0,1,\ldots,N-1$$

Inverse DFT : $$x(n) = \sum_{k=0}^{N-1} X(k)\, W_N^{nk} \qquad n = 0, 1,\ldots,N-1$$

where $$W_N = exp(-2\pi j/N)$$

The implementation of the above pair, as it is written would require $4N^2$ real multiplications and $(4N-2)$ real additions. Many methods have been developed to improve the efficiency of the computation of the DFT, exploiting the symmetry and periodicity properties of W_N^{kn};

1.Symmetry; $$W_N^{k[N-n]} = W_N^{-kn}$$

2.Periodicity; $$W_N^{kn} = W_N^{k(N+n)}$$

Goertzel algorithm [10,12]:

Goertzel algorithm uses the periodicity property of the sequence W_N^{nk} to reduce computation, by.expressing the DFT as a linear filtering operation.

The spectrum can be expressed as $X(k) = y_k(n)|_{n=N}$ where $y_k(n)$ is the convolution of the input sequence $x(n)$ of length N with a filter that has an impulse response

$$H_k(z) = (1 - W_N^k z^{-1}) / (1 - 2cos(2\pi k/N)z^{-1} + z^{-2}).$$

The system can be described by the following difference equations

$$v_k(n) = 2cos(2\pi k/N)\, v_k(n-1) - v_k(n-2) + x(n) \qquad (5.3)$$

$$y_k(n) = v_k(n) - W_N^k\, v_k(n-1) \qquad (5.4)$$

Eq.(5.3) is calculated for $n=0, 1,\ldots,N-1$, but Eq.(5.4) is computed only once at time $n=N$. When the input data is complex, this algorithm requires $2n-1$ real multiplications and $4n-1$ real additions for each output component computed.

Steps in the implementation of sequentia algorithm are:

1. Initialize the parameters.
2. Compute Eqs.5.3 and 5.4, to calculate $X(k)$ for $k=0$.
3. Repeat step 2 for other values of k, till $k=N-1$.
4. Store the results.

Computation of $X(k+1)$ does not depend in any way on the value of $X(k)$ and computation of $X(k)$ for each k needs all samples of the input sequence. So they can be calculated concurrently on different processors. Hence the algorithm for any processor in a transputer network can be given as;

1. Receive N input values.
2. Compute N/p output values by using Eqs.5.3 and 5.4.
3. Send N/p output values.

The root processor reads input values and sends them to the first processor. The first processor sends them to all processors, which compute and send back computed values to the first processor, which in turn sends them to the root processor.

5.4. Fast Fourier transform (FFT):

The fast Fourier transform uses the periodicity and symmetry properties of the sequence W_N^{nk} to reduce computation. All FFT algorithms are based on the fundamental principle of decomposing the computation of the discrete Fourier transform of a sequence of length N into successively smaller discrete Fourier transforms. The manner in which this principle is implemented leads to a variety of algorithms. Here we consider ***decimation in frequency*** (DIF) class of algorithms. In this class of algorithms, the sequence of discrete Fourier transform coefficients $X[k]$ is divided into smaller subsequences.

Cooley–Tukey radix 2 FFT algorithm [11,13,14]:

N is a power of 2. The decimation of the output sequence is accomplished by dividing $X(k)$ into two equations, one that computes even output samples and one that computes odd output samples.

Even values of $X(k)$ are calculated by

$$X(2r) = \sum_{n=0}^{N/2-1} [x(n)+x(n+N/2)]\, W_{N/2}^{nr},\ r=0,1,\ldots,N/2-1. \tag{5.5}$$

Odd values of $X(k)$ are calculated by

$$X(r+1) = \sum_{n=0}^{N/2-1} [x(n)-x(n+N/2)] W_{N/2}^{nr}\, W_{N}^{n},\ r=0,1,\ldots,N/2-1 \tag{5.6}$$

$X(2r)$ and $X(2r+1)$ are $N/2$ point DFTs performed on the sum and difference of the two halves of the input sequence. In Eq.5.6, the difference of the two halves is multiplied by a factor W_N^n, known as *twiddle factor*. Each of the two $N/2$ – point DFTs, can in turn, be divided into two $N/4$ point DFTs in the same way. The process is continued till all the factors are 2. Fig.5.1 shows the complete flowgraph for 8–point DIF FFT. The computation requires $(N/2)\log_2 N$ complex multiplications and $N \log_2 N$ complex additions. The basic computational structure is a 2 point DFT known as butterfly,as shown in Fig.5.2.

By rearranging the branch transmittances in each stage, different forms of flowgraphs can be obtained. One of them, called the constant–geometry version is given in Fig.5.3. This flowgraph is used in this thesis because it allows the same data shuffling for all the stages.

Steps in the implementation of sequential algorithm are:

1. Initialize the parameters.
2. Perform $N/2$ butterfly calculations for all the input values in the stage.
3. Repeat step 2 for all the stages in FFT flowgraph.
4. Unscramble the output values, and store the results.

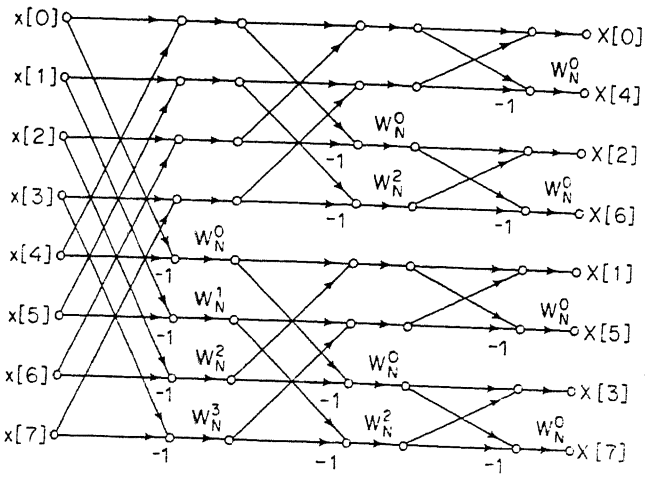

Fig.5.1: Flow graph of DIF decomposition of 8 point DFT computation

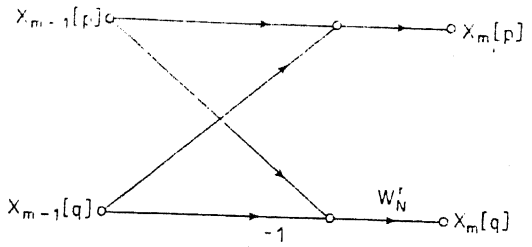

Fig.5.2: Flow graph of a radix 2 DIF butterfly computation

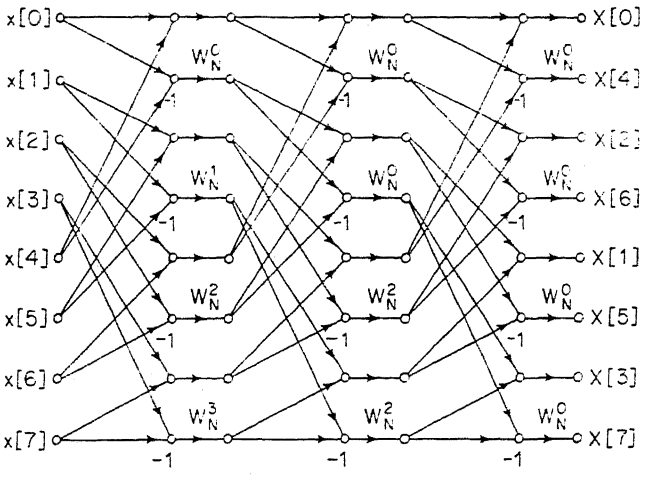

Fig.5.3: Rearrangement of Fig.5.1 (Constant geometry version)

Step 2 has parallelism. If N is the size of the input $x(n)$, $N/2$ butterflies will be performed in each stage. The inputs to the butterflies are

$x(i)$ and $x(i+N/2)$ where $i = 0, 1,...,(N/2)-1$

The inputs to each butterfly are independent of each other. So they can be calculated concurrently on different processors. Hence the algorithm for any processor in a transputer network can be given as;

1.
   a. Receive N/p input values.
   b. Compute $N/2p$ butterflies.
   c. Send N/p output values
2. Repeat step 1 for all the stages.

The root processor reads input values and sends them to the first processor. The first processor distributes them to different processors, which compute and send back computed values to the first processor. The first processor rearranges the received values and if all stages are not completed it again distributes them to different processors, otherwise it sends to the root processor.

Cooley–Tukey radix 4 FFT algorithm [11,15]:

N is a power of 4. Here N point sequence is broken into four $N/4$ point sequences, each $N/4$ sequence is, in turn, broken into four $N/16$ point sequences and so on, until only four point DFTs are left. The four point DFT is core calculation (butterfly) of radix 4 (Fig.5.4).

The four sub sequences of the output are computed by

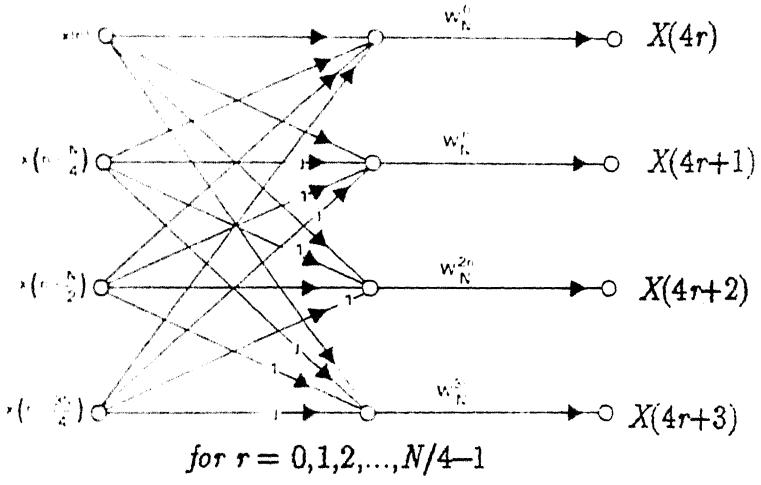

Fig.5.4: Flow graph of a radix 4 DIF butterfly computation

$$X(4r) = \sum_{n=0}^{N/4-1} [\, x(n)+x(n+N/4)+x(n+N/2)+x(n+3N/4)\,]\, W_N^{0}$$

$$X(4r+1) = \sum_{n=0}^{N/4-1} [\, x(n)-jx(n+N/4)-x(n+N/2)+jx(n+3N/4)]\, W_N^{n}$$

$$X(4r+2) = \sum_{n=0}^{N/4-1} [\, x(n)-x(n+N/4)+x(n+N/2)-x(n+3N/4)\,]\, W_N^{2n}$$

$$X(4r+3) = \sum_{n=0}^{N/4-1} [\, x(n)-jx(n+N/4)-x(n+N/2)-jx(n+3N/4)\,]\, W_N^{3n}$$

for $r=0,1,\ldots,N/4-1$.

Steps in the implementation of Radix 4 sequential algorithm are same as that of the Radix 2 algorithm except for the number of butterflies which is $N/4$ in this case. Hence the same steps as given in the case of Radix 2 can be used in the execution of the algorithm on a transputer network.

5.5. Two dimensional Fourier transform [16,17]:

The properties of signals and systems that are applicable to one dimensional case are equally applicable to two dimensional case.

The DFT pair of a 2–D sequence $x(m.n)$, $m = 0,1,2,\ldots,M-1$, $n = 0,1,2,\ldots,N-1$, is defined as

DFT : $$X(u,v) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} x(m,n)\, exp\,[-j2\pi\,(um/M + vn/N)]$$

where $u = 0,1,2,\ldots,M-1$; $v = 0,1,2,\ldots N-1$.

Inverse DFT : $$x(m,n) = (1/MN) \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} X(u,v)\, exp\,[j2\pi\,(um/M + vn/N)]$$

where $m = 0,1,2,\ldots,M-1$; $n = 0,1,2,\ldots N-1$.

Two dimensional Fourier transform is viewed as two successive one dimensional transforms.

$$X(u,v) = \sum_{m=0}^{M-1} exp[-j2\pi um/M] \{ \sum_{n=0}^{N-1} x(m,n)\, exp[-j2\pi vn/N] \}$$

If the value of m is considered as a parameter, the summation inside the brackets, { }, in the above equation has the form of a 1–D DFT. This means that an FFT algorithm can be used to perform this computation for each value of m, a total of M FFTs. These computations will produce a matrix of values, say $g(m,v)$, whose indices are m and v. For each value of v there is a function of m which can be called as $g_v(m)$. The equation can now be written as,

$$X(u,v) = \sum_{m=0}^{M-1} g_v(m)\, exp[-j2\pi um/M]$$

This set of computations also has the form of a DFT and can be computed using a separate FFT for each value of v, a total of N FFTs. The total number of computations is $NM\log_2 NM$.

Steps in the implementation of the sequential algorithm are:

1. Initialize the parameters.
2. Compute the FFT (inplace computation) of each row of data, having N points.
3. Compute the FFT (inplace computation) of each column of data, having M points.
4. Store the results.

Steps 2 and 3 have parallelism. Computation of the DFT of any row (or column) of data is independent of the computation of the DFT of any other row (or column) of data. Hence they can be calculated concurrently on different processors. The algorithm for any processor in a transputer network can be given as;

1

a. Receive M/p rows of input data, each row having N points.

b. Compute the FFT of each row of input data, using sequential algorithm.

c. Send M/p rows of transformed data.

2

a. Receive N/p column of input data, each column having M points.

b. Compute the FFT of each column of data, using sequential algorithm.

c. Send N/p columns of transformed data.

The root processor reads input values and sends them to the first processor. The first processor distributes them row wise to different processors, which compute and send back computed values to the first processor. The first processor redistributes the received values column wise to different processors, which compute and send back values to the first processor, which in turn sends it to the root processor.

5.6. Two dimensional convolution [16,17]:

Two dimensional convolution relation between two signals, x of size $(N \times N)$ and h of size $(M \times M)$ is given by:

$$y(r,s) = \sum_{m=0}^{N-1} \sum_{n=0}^{N-1} x(m,n)\, h(r-m, s-n)$$

$$r = 0,1,\ldots,P;\quad s = 0,1,\ldots,P;\quad \text{where } P=(M+N-1).$$

2–D convolution can also be calculated using 2D FFT and inverse 2D FFT. The *convolution theorem* states that time domain convolution is equivalent to multiplication in the frequency domain. The convolution equation above can be written in the frequency domain as:

$$Y(u,v) = H(u,v)^{*}X(u,v)$$

where Y,H,X are frequency domain representations of y,h,x respectively. The sequence y can be obtained by the inverse Fourier transform of Y. For the convolution theorem to be valid the same size DFT must be used for each transformation and to avoid *wrap around* effect (i.e overlapping of the convolution result of one period with that of the succeeding period) due to the periodicity of the DFT, it is necessary to pad the larger of the two signals with a number zeroes in each dimension equal to the size of the smaller of the two signals. This is called *zero padding*. The size of the signal must be a power of 2 (for a radix 2 FFT) in length on each side. This may also require some extra zero padding.

Steps in the implementation of the sequential algorithm are:

1. Initialize the parameters.
2. Compute $X(u,v)$, the 2D FFT of the signal $x(m,n)$.of size $(P\times P)$.
3. Compute $H(u,v)$, the 2D FFT of the signal $h(m,n)$.of size $(P\times P)$.
4. Compute $Y(u,v)$, the pointwise multiplication of $X(u,v)$ and $H(u,v)$.
5. Compute $y(m,n)$, the inverse 2D DFT of $Y(u,v)$ of size $(P\times P)$.
6. Store the results.

The number of the computations required is given by:

$$2N^{2} + 8N^{2}\log(N)$$

We have seen that the computation of the 2D FFT can be parallelized. Hence step 2 can be computed concurrently on different processors, and so steps 3 and 4. The algorithm for any processor in a transputer network can be given as;

1.

a. Receive P/p rows of $x(m,n)$.

b. Compute the FFT of P/p rows of data.

c. Send P/p rows of transformed data.

d. Receive P/p columns of $x(u,n)$.

e. Compute the FFT of P/p columns of data.

2. Repeat the step 1 for $h(m,n)$.

3. Compute the pointwise multiplication of the values obtained in steps 1 and 2.

4. Compute the inverse FFT of P/p columns obtained in step 3.

5. Send P/p columns of $Y(u,n)$ obtained in step 4.

6. Receive P/p rows of $Y(u,n)$.

7. Compute the inverse FFT of P/p rows of $Y(u,n)$.

The root processor reads the two signals to be convolved and sends them to the first processor. The first processor distributes one signal row wise to different processors, which compute and send back values to the first processor. The first processor redistributes the received values column wise to different processors, which do computation. Now the first processor distributes another signal row wise to different processors, which compute and send back values to the first processor, which redistributes the received values column wise to different processors, which do computation and send back values to the first processor. The first processor redistributes the received values row wise to different processors, which do computation and send back computed values to the first processor, which in turn sends them to the root processor.

5.7. Walsh Hadamard transform (WHT) [18]:

The discrete Walsh–Hadamard transform of a function $f(x)$ of length N, denoted by $F(w)$, is given by

$$F_i(w) = \sum_{j=0}^{N-1} (-1)^{\sum_{k=0}^{N-1} i_k j_k} x_j$$

where i_k, j_k are the k th bits when i, j are represented as N bit numbers.

The array formed by the transformation kernel has rows and columns which are orthogonal. The inverse kernel is identical to the forward kernel except for the constant multiplication factor of $1/N$. The kernels consist of 1 and -1 only, so the operations involved are additions and subtractions only. The WHT is a matrix–vector multiplication similar to the DFT. Similar to DFT the Walsh–Hadamard has a fast algorithm called *fast Walsh–Hadamard* transform.

The structure of fast Walsh–Hadamard algorithm is similar to the constant geometry FFT algorithm. It has $\log_2 N$ stages. The computation is identical in all stages. The data shuffling between adjacent stages is also identical. Fig.5.5 shows FWHT flowgraph for N=16. The circles represent an addition and subtraction operations. Let us call the basic computational structure which is a 2 point WHT as *Walsh butterfly*.

Steps in the implementation of sequential algorithm are:

1. Initialize the parameters.
2. Perform $N/2$ Walsh butterfly calculations for all the input values in the stage
3. Repeat step 2 for all the stages in WHT flowgraph.
4. Store the results.

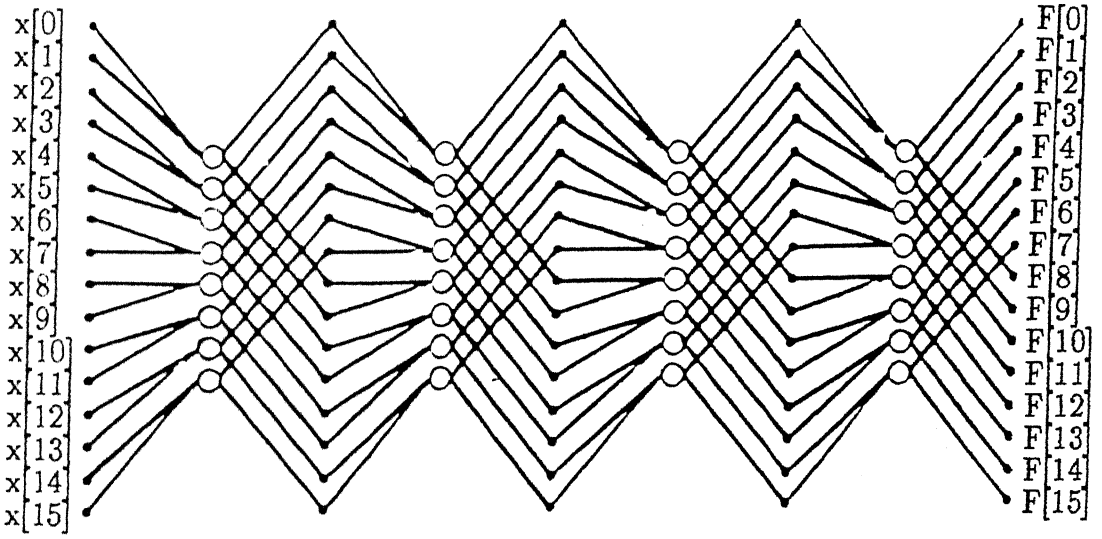

Fig.5.5: Flow graph of 16 point WHT computation

Step 2 has parallelism. If N is the size of the input $x(n)$, $N/2$ Walsh butterflies will be performed in each stage. The inputs to the butterflies are

$x(i)$ and $x(i+1)$ where $i = 0, 2, 4, \ldots, (N-2)$

The inputs to each butterfly are independent of each other. So they can be calculated concurrently on different processors. Hence the algorithm for any processor in a transputer network can be given as;

1.
   a. Receive N/p input values.
   b. Compute $N/2p$ Walsh butterflies.
   c. Send N/p output values.
2. Repeat step 1 for all the stages.

The root processor reads input values and sends them to the first processor. The first processor distributes them to different processors, which compute and send back computed values to the first processor. The first processor rearranges the received values and if all stages are not completed it again distributes them to different processors, otherwise it sends the values to the root processor.

Chapter 6
RESULTS AND DISCUSSIONS

The performance of a parallel algorithm is an important consideration. Now we shall see the performance of the algorithms discussed in Chapter 5. The execution times of the sequential and parallel versions of the algorithms are measured for different sizes of input data on a network of 4 transputers (of type T800). Speedup factor and efficiencies which are defined in Chapter 1 are computed. Similarly the execution times are measured for one value of input data on networks of 2,4,8 transputers (of type T414). Speedup factors are computed. These are given in the tabular form for different algorithms.

6.1

Results:

a.: Results on network of 4 transputers (of type T800):

Table 6.1: Results of matrix multiplication

Sl.No	Data Size	Execution time(ms)		Speedup	Efficiency %
		Sequential	Parallel		
1	8 x 8	4.29	3.07	1.4	35.00
2	16 x 16	33.92	19.20	1.77	44.25
3	32 x 32	267.07	88.00	3.03	75.75
4	64 x 64	2147.46	598.40	3.59	89.75
5	128 x128	17118.53	4516.49	3.79	94.75

Table 6.2: Results of convolution

Sl.No	Data Size	Execution time(ms) Sequential	Parallel	Speedup	Efficiency %
1	8	0.26	0.58	0.44	11.00
2	16	0.96	1.28	0.75	18.75
3	32	3.84	2.88	1.33	33.25
4	64	15.17	7.49	2.03	50.75
5	128	60.54	21.95	2.76	69.00
6	256	248.51	75.65	3.29	82.25
7	512	992.77	280.90	3.53	88.25
8	1024	3968.00	1104.58	3.59	89.75

Table 6.3: Results of Goertzel algorithm

Sl.No	Data Size	Execution time(ms) Sequential	Parallel	Speedup	Efficiency %
1	8	0.83	0.45	1.86	46.50
2	16	2.62	1.02	2.56	64.00
3	32	8.90	2.88	3.08	77.00
4	64	32.51	9.41	3.46	86.50
5	128	123.97	33.22	3.73	93.25
6	256	483.46	124.16	3.89	97.25
7	512	1908.80	485.50	3.93	98.25
8	1024	7610.94	1935.04	3.93	98.25

Table 6.4: Results of FFT (radix 2)

Sl.No	Data Size	Execution time(ms) Sequential	Parallel	Speedup	Efficiency %
1	8	0.51	0.51	1.00	25.00
2	16	1.34	1.02	1.31	32.75
3	32	3.20	2.11	1.51	37.75
4	64	7.49	4.54	1.65	41.25
5	128	17.09	9.92	1.72	43.00
6	256	38.66	23.30	1.66	41.50
7	512	85.57	52.22	1.65	41.25
8	1024	187.90	113.99	1.65	41.25

Table 6.5: Results of FFT (radix 4)

Sl.No	Data Size	Execution time(ms) Sequential	Parallel	Speedup	Efficienc %
1	16	1.28	0.64	2.00	50.00
2	64	6.46	2.88	2.24	56.00
3	256	31.30	14.34	2.18	54.50
4	1024	144.51	68.29	2.12	53.00

Table 6.6: Results of fast Walsh–Hadamard transform

Sl.No	Data Size	Execution time(ms) Sequential	Parallel	Speedup	Efficiency %
1	8	0.06	0.32	0.20	5.00
2	16	0.26	0.64	0.40	10.00
3	32	0.58	1.34	0.43	10.75
4	64	1.34	2.88	0.47	11.75
5	128	3.07	6.44	0.47	11.75
6	256	7.23	14.4	0.50	12.50
7	512	16.38	32.00	0.51	12.75
8	1024	36.42	71.36	0.51	12.75

Table 6.7: Results of 2–D FFT

Sl.No	Data Size	Execution time(ms) Sequential	Parallel	Speedup	Efficiency %
1	8 x 8	9.66	4.29	2.25	56.29
2	16 x 16	46.02	18.94	2.43	60.75
3	32 x 32	205.7	80.70	2.55	63.75
4	64 x 64	904.96	343.36	2.64	66.00
5	128 x 128	3917.63	1442.43	2.72	68.00

Table 6.8: Results of 2–D convolution

Sl.No	Data Size	Execution time(ms) Sequential	Parallel	Speedup	Efficiency %
1	4 x 4	31.87	11.26	2.83	70.75
2	8 x 8	150.59	49.86	3.02	75.51
3	16 x 16	670.08	214.91	3.12	78.00
4	32 x 32	2907.71	925.44	3.14	78.50
5	64 x 64	12534.14	3953.54	3.17	79.50

b.: Results on networks of 2,4,8 transputers (of type T414):

Table 6.9: Execution times

Sl.No	Algorithm	Execution time(ms) Sequential	2 Node	4 Node	8 Node
1	Matrix multiplication (32×32)	2489.66	1253.44	641.66	345.73
2	Convolution (128 points)	1211.14	617.88	326.66	215.74
3	Goertzel algorithm (128 points)	4955.78	2500.54	1257.09	638.59
4	FFT (rad 2) (256 points)	455.94	257.41	149.82	105.60
5	FFT (rad 4) (256 points)	408.7	216.32	125.50	
6	2–D FFT (32×32)	3331.39	1713.60	901.50	498.50
7	2–D Convolution (16×16)	11292.09	5843.33	3228.74	1745.15

Table 6.10: Speedup factors

Sl.No	Algorithm	Speedup factor 2 Node	4 Node	8 Node
1	Matrix multiplication	1.98	3.88	7.20
2	Convolution	1.96	3.70	5.60
3	Goertzel algorithm	1.98	3.94	7.76
4	FFT (rad 2)	1.77	3.04	4.32
5	FFT (rad 4)	1.89	3.26	
6	2–D FFT	1.94	3.69	6.68
7	2–D Convolution	1.93	3.49	6.47

Discussions:

For matrix multiplication, convolution, Goertzel algorithm, 2–D FFT, and 2–D convolution the speedup is observed to be increasing in an approximately linear manner with the data size.

As a general rule, speedup is a non decreasing function of the input data size, but in the case of radix 2 and radix 4 algorithms the speedup increases and then decreases. (radix 2 algorithm has a maximum value of 1.72 for data size 128, radix 4 algorithm has a maximum value of 2.24 for data size 64). The reason for this behaviour may be, there is a right balance between the computation time and the communication time at that point and afterwards increase in the communication time may not be proportional to increase in the computation time.

Speedup is very poor (less than 1) in the case of fast Walsh–Hadamard transform algorithm. The reason for inefficient parallel algorithm is, the operations involved in WHT are only additions and subtractions, so the computation time is very less compared to the communication time.

When we parallelize DFT (Goertzel algorithm) we obtain better results, than the results obtained by parallelizing FFT (radix 2, radix 4). The reason may be that the fast algorithms have been designed with sequential processors in mind, and hence the parallelization of these algorithms might be incapable of producing very satisfactory results.

The above results indicate that the transputer based parallel processing systems appear to be advantageous in problems involving considerable computational activities.

6.2. Suggestions for further work:

Transputers with more number of links are likely to be available. Using these algorithms can be implemented more efficiently as communication overheads will be less. Fast Fourier transform algorithms may be designed with parallel processors in mind. In this direction, FFTs may be designed using residue number system, the most relevant characteristic of which is its carry free arithmetic.

[13] C. S. Burrus and T. W. Parks. *DFT/FFT and Convolution Algorithms.* John Wiley and Sons, 1985.

[14] R. Taylor. *Signal Processing with Occam and the Transputer.* IEE Proceedings, Pt F, No 6, October 1984.

[15] L. R. Rabiner and B. Gold. *Theory and Application of Digital Signal Processing.* Prentice Hall, 1975.

[16] E. O. Brigham. *The Fast Fourier Transform and its Applications.* Prentice Hall, 1988.

[17] P. M. Embree and B. Kimble. *C Language Algorithms for Digital Signal Processing.* Prentice Hall, 1991.

[18] R. Gonzalez and P. Wintz. *Digital Image Processing.* Addison–Wesley, 1977.

EE-1992-M-SHR-TRA